

TOOLS FOR ENFORCING SOFTWARE CODING STANDARDS

by

DEEPJYOTI KAKATI

CSA
1997
M
KAK

TH
CSE/1997/4
R1236



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JANUARY 1997

Tan

TOOLS FOR ENFORCING SOFTWARE CODING STANDARDS

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by
Deepjyoti Kakati

to the
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
February, 1997

CENTRAL LIBRARY
I. I. T., KANPUR


No. A. 123148

CSE-1997-M-KAK-T00

3.297
11/6/97

CERTIFICATE

Certified that the work contained in the thesis entitled "Tools for enforcing software coding standards", by "Deepjyoti Kakati(9511105)", has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.



Dr. S.K. Aggarwal
Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

February, 1997

*dedicated to
my late grandmother*

Acknowledgements

I am grateful to my thesis supervisor, Dr. S.K.Aggarwal who gave me the opportunity to work in the interesting fields of compilers and software tools. He was always ready to discuss and give insights into the problems encountered during the thesis. I would have been unable to complete this thesis but for his knowledge and encouragement.

I would like to thank the staff of CSE lab for keeping the machines in order and extending help where necessary.

I thank all my friends here who helped me in every possible way. Thanks to Nigam, Sameer, Kapil, Kommu, Praveen, Negi, Nabanjan, Sharad and Ambarish for their help and enjoyable company. Special thanks to Sameer for proof-reading my thesis report and to Nigam for his moral support.

I acknowledge my gratitude to my parents and sister for their love and support.

–Deepjyoti Kakati

Abstract

This thesis explores the techniques by which Quality Assurance tools can be created for enforcing software coding standards. Two tools were created to enforce coding standards, one for C and the other for Ada. Based on the knowledge gained from constructing these two tools, a Generator Tool was made that generates a skeleton front-end from any language grammar supplied by the user. This skeleton front-end can be manually enhanced to make it a coding standard enforcing tool. The possibility of using syntax-based editors for enforcing coding standards was also studied.

Contents

Acknowledgements	iv
Abstract	v
1 Introduction	1
1.1 Motivation	1
1.2 Goal of the thesis	3
1.3 Tools developed	3
1.4 Related work	5
1.4.1 Constraint enforcing tools	5
1.4.2 Generator tools	6
1.5 Organization of the thesis	6
2 Rationale for using coding standards	8
2.1 Classification of constraints	8
2.2 Examples of constraints	9
2.2.1 Syntax constraints	9
2.2.2 Semantic constraints	10
2.2.3 Dataflow constraints	11
3 Implementation of coding standard enforcers	13
3.1 Specification of coding standards	13
3.1.1 Method of specifying coding standards	13
3.1.2 Coding standard header files	14

3.1.3	Format of the constraint selection file	14
3.1.4	Output from makestd	16
3.1.5	Integration with the constraint enforcing system	17
3.1.6	Introducing new constraints	17
3.2	Enforcement of coding standards	20
3.2.1	Technique for enforcing constraints	20
3.2.2	Examples of constraint enforcement	22
3.2.3	Generation of warnings	23
3.2.4	Automatic restructuring to remove violations	25
4	Generation of coding standard enforcers	26
4.1	Benefits of a Generator Tool	26
4.2	createparse—a tool for partial automation	27
4.2.1	Format of the language specification file	27
4.2.2	Building the complete front-end	29
5	Implementation details of the Tools	32
5.1	A note on Flex and Bison	32
5.2	Languages and development tools used	33
5.3	Statistics about the software developed	34
5.4	Development of checkc—the C constraint enforcer	34
5.5	Development of checkada—the Ada constraint enforcer	35
5.6	Development of createparse—the Generator Tool	35
6	Conclusion and further work	36
6.1	Comparisons with existing tools	37
6.2	Limitations	37
6.3	Further work	38
A	List of constraints enforced by checkc	39
B	List of constraints enforced by checkada	43
C	User manuals	50

List of Figures

1	Usage of coding standards in software organizations	4
2	Action of the constraint specification process	18
3	Schematic of constraint specification and system integration	19
4	Action of the constraint enforcing tools	24
5	Action of createparse and genbison	30
6	Generation of the skeleton front-end	31

Chapter 1

Introduction

1.1 Motivation

When programming in the large, the main goal is not only to obtain a working program, but also to produce rugged software that can be modified, updated and debugged easily.

Organizations developing large software define their coding standards as a mechanism to filter out what they may consider “undesirable” features in their delivered code. Coding standards are composed of constraints that are placed upon the code. Constraints are assertions about using or not using some feature, or using features in certain ways. By putting constraints on features of code which are considered “undesirable”, organizations can ensure a better quality of code for their needs. Programmers working in such organizations have to produce code that conforms with the set standard.

Because of the size and complexity of modern projects and strict deadlines in which they have to be delivered, it is usually not possible for project managers to manually check the code for conformance to the coding standard. Certain standard violations are very difficult and time-consuming to detect manually and may slip through a manual check. Also it is difficult for programmers to continuously keep track of components in a comprehensive coding standard while coding.

Given this situation, organizations need tools that allow them to define and enforce coding standards. Programmers can use the tools to check their code for conformance and warn about violations.

The coding standard enforcing tools operate in two ways:

Interactive mode: the coding standards are enforced when a user types the program, by disallowing entry of forbidden constructs or immediately annotating the code with warning(s). The warning(s) vanish when the violation is rectified. This category of tools falls in the realm of enhanced syntax-directed editors. They can work even on incomplete programs

Batch mode: A user types out the complete program and submits it to the tool, which then prints out warnings after analyzing the entire program. The user can use this feedback provided to modify the code and remove the violations

From an organization's point of view, the standard enforcing tools should have the following features:

Flexibility: they are not intended to impose a specific coding style. A large enough set of constraints should be available so that the choice of constraints is not unduly restrictive

Extensibility: well documented so they can be modified and enhanced if necessary to support future needs

From a programmer's point of view, the tools should be:

Easy to use: they should be as easy to use as a compiler

Informative: the warnings produced should be informative and help the programmer easily locate the causes of the warnings

It is also desirable that similar tools be available for enforcing standards for as many languages as an organization works in. Similarity is desired in the process of creating the coding standards, method of using the enforcers and output from the enforcers. This lessens the learning time for new users.

Figure 1 shows how coding standards are used by software organizations.

Current constraint enforcing tools have certain drawbacks which make them somewhat unsuitable for enforcing coding standards. `Lint`[12] does not allow fine-grained control over what checks to perform. It's source is also not available in the public-domain and hence users cannot modify it for their own needs. `LCLint`[7] requires the user to insert additional comments in special format into the code. Both of these operate in batch mode.

1.2 Goal of the thesis

The goal of the thesis is to develop techniques and tools to enforce software coding standards. The techniques are required to be sufficiently general to be useful in building coding standard enforcing programs for most languages. To prove this, the same techniques will be applied to build tools for more than one language. The developed tools will have to possess the desirable features mentioned earlier. The construction of a Generator Tool will also be attempted after building constraint enforcers for two common languages—C and Ada.

1.3 Tools developed

The tools developed during the thesis are:

`checkc` a coding standard enforcer for C. The user can turn on any or all of the constraint checks. It operates in batch mode

`checkada` a coding standard enforcer for Ada. The user can turn on any or all of the constraint checks. It operates in batch mode

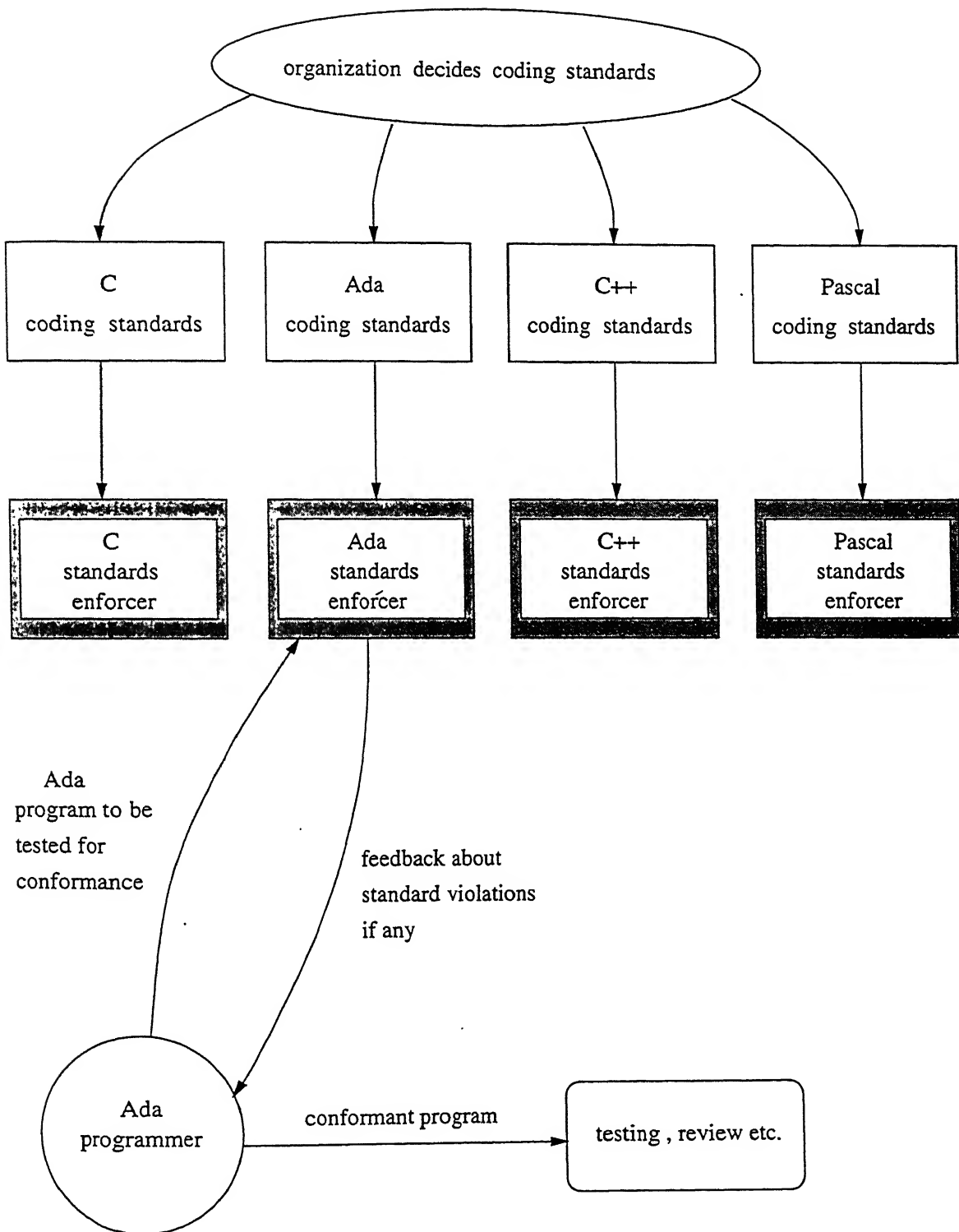


Figure 1: Usage of coding standards in software organizations

`createparse` a Generator Tool to generate a skeleton front-end given the grammar of the input language. It builds a skeleton lexer specification, a full parser specification and code to build and traverse the parse-tree of the input

1.4 Related work

Work on developing software Quality Assurance tools is being done in both academic and commercial environments. A few such efforts are briefly described here.

1.4.1 Constraint enforcing tools

Open Systems Portability Checker(OSPC) made by Knowledge Software Co. of UK[20] checks applications written in C for conformance to a wide variety of standards. It can be used for quality-assurance and standards-adherence testing among other things.

Ada-Assured[23] is a smart source editor marketed by Grammatech Inc. of USA. It is intended to help produce high quality Ada95 and Ada83 programs. It consists of a language-sensitive editor, style standards enforcer, browser, pretty-printer and quality assurance tool.

Lint is a C program checker that is available on all Unix systems. It attempts to detect features of C programs that are likely to be bugs, non-portable or wasteful. It also checks type usage more strictly than the C compilers.

LCLint from the Systematic Program Development Group at MIT is a tool for statically checking C programs. It can be used as a better lint with minimal effort. If additional effort is invested in adding annotations to programs, LCLint can perform stronger checks than can be done by any standard lint. Some of the checks that can be done are violations of information hiding, inconsistent use of global variables, memory management errors, dangerous data-sharing or unexpected aliasing, using

possibly undefined storage or returning storage that is not completely defined, dereferencing a possibly null pointer, dangerous macro implementations or invocations, violations of customizable naming conventions, program behaviour that is undefined because it depends on order of evaluation, likely infinite loops, fall through cases, incomplete logic, statements with no effect, ignored return values, unused declarations and exceeding certain limits. [15] is the user-manual for LCLint.

1.4.2 Generator tools

The Synthesizer Generator[5] from Grammatech Inc.[23] is a tool for generating structure-oriented environments. It receives a specification and produces an editor that supports syntax-directed editing, incremental compilation, structured interpretation and debugging. A specification of a programming language consists of an attributed grammar which is used to analyze the lexical, syntactical and semantic level of the text. It has been used in a wide range of applications, for example to create a Pascal environment with full static-semantic checking, code-generation and interpretation, as well as editors for C, Ada, Fortran77 and SQL. Many other editors have been created for software and hardware specification, verification and proof checking.

1.5 Organization of the thesis

Chapter 2 discusses the necessity of coding standards in constraining the use of languages and classifies types of constraints that are possible

Chapter 3 describes how the constraints are specified to the coding standard enforcing tools and how they are enforced

Chapter 4 discusses the issues involved in generating coding standard enforcers automatically

Chapter 5 discusses the implementation of the C and Ada standards enforcers, `checkc` and `checkada` and the Generator Tool `createparse`

Chapter 6 gives the conclusions, comparisons with existing tools, limitations and scope for further work

Appendices list the constraints supported for C and Ada. User manuals are also provided for all the tools developed

Chapter 2

Rationale for using coding standards

Many programming languages have features that are tricky to use and hard to understand. Organizations engaged in large software development restrict the use of such features by using a subset of the full language, so that the code produced is easy to debug and maintain. This is worthwhile because a large proportion of the time spent on a project is in debugging. Almost all the “undesirable” language features are quite useful in certain situations but organizations want to carefully control their usage by defining their own coding standards to constrain use of such features.

Constraints are placed on language features to improve various qualities of the code like readability, portability, understandability and maintainability.

2.1 Classification of constraints

Constraints which make up coding standards can be classified into three types.

Syntax constraints: these concern the use of syntax features. For example, *goto statements are not to be used*

Semantic constraints: these require information on type. For example, *floating point numbers are not to be used in comparison expressions*

Dataflow constraints: these need information on definition and use of identifiers. For example, *All variables should be initialized prior to being used*

2.2 Examples of constraints

Some examples of the different types of constraints that are possible are given below along with the rationale for using them. All the examples are from the C language. A full listing of the constraints that are supported by `checkc` (for C) and `checkada` (for Ada) are given in Appendices A and B.

2.2.1 Syntax constraints

Check if statements with missing else parts: although it may be perfectly reasonable in certain contexts, an if-else chain with no final else may indicate missing logic or forgetting to check some error condition

Check if the body of certain statements is not block: readability is improved if if, while, do-while and for statements have bodies that are blocks

Check if header file names clash with that of common system header files: potential confusion for the maintainer is avoided

Check for similar looking names: removes a source of potential confusion for developer and maintainer

Set the significant length for external names: some compilers have limits i.e all external names should be significant within some number of characters

Set the maximum number of fields that a structure may have: some compilers have small limits on this number. this constraint can be used to take care of this concern. ANSI C has a limit of 127

2.2.2 Semantic constraints

Conditional test should not be assignment: this usually indicates a careless typing error by the developer. For example, a programmer may type

```
if ( counter = COUNTER_LIMIT ) {  
    ....  
}
```

rather than

```
if ( counter == COUNTER_LIMIT ) {  
    ....  
}
```

Characters not allowed to index arrays: characters have a limited range and will not be able to index large arrays. Example:

```
int    newarr[1000];  
char   arrindex;  
arrindex = 400;      /* wrong! cant hold this value */  
varx = newarr[ arrindex ]; /* incorrect assignment */
```

No float or double variable allowed in conditional expressions: errors can be introduced in using `==` and `!=` operators to compare floating point types. Expressions such as $(float_expr_1 == float_expr_2)$ will seldom be satisfied due to rounding errors. For example, if we try to evaluate $10^{50} + 812 - 10^{50} + 10^{55} + 511 - 10^{55} = 812 + 511 = 1323$, most computers will produce 0 regardless of whether one uses float or double

Warn about same name reused in different scopes: this removes a potential cause of errors. Also the masked off name(s) cannot be used directly in the inner scope unless we have another variable pointing to them. Example:

```

int varx = 10;
{
    char varx = 'c';
    /* user may try to use varx as an integer here */
}

```

All non-extern global variables should be static: this will restrict the scope of variables to the file where they are declared and prevent name clashes which will cause linker errors

Disallow expressions with side-effects: expressions having side-effects always complicate the understanding of the code and makes maintainance difficult. Example:

```

while ( i++ > j-- ) {
    ...
}

```

Check for assignments between signed and unsigned variables: this warns about assignments between signed and unsigned variables. An unsigned type cannot represent all signed values correctly. Example:

```

signed int a;
unsigned int b;
a = -10;
b = a;    /* wrong! information is lost */

```

2.2.3 Dataflow constraints

Check for dynamic storage used after freeing: compilers do not check this. The state of storage after freeing is undefined and will probably lead to a runtime error or even worse, silently change the program behaviour. Example:

```

char *cptr;
char a;
cptr = (char *)malloc(800*sizeof(char));
... /* cptr used here */
free( cptr );
a = *(cptr+10); /* wrong! state of cptr is undefined */

```

Check for dynamic storage not freed prior to scope exit: it is always good to free explicitly rather than relying upon anything else. Sometimes even when out of scope, a variable's storage may be accessible due to implementation errors in the compiler

```

{
char *cptr;
cptr = (char *)malloc(500*sizeof(char));
... /* cptr used here but not freed */
}
a = *(cptr+9); /* wrong! cptr may still be accessible */

```

Check for paths in functions having no return statements: it is important to check this in case the function returns a non-void type but the compiler does not check this

Chapter 3

Implementation of coding standard enforcers

The two issues in implementing tools to enforce coding standards are

1. Specification of coding standards to the enforcers
2. Enforcement of the coding standards

3.1 Specification of coding standards

3.1.1 Method of specifying coding standards

The module where a user specifies the coding standards to be enforced is kept separate from the actual tools that enforce the standards.

The method used to take the users choice of the coding standard is that the user submits a selection file and a program `makestd` reads this input file and generates some C code that encodes the users choices. The output of `makestd` is compiled and linked with the object files of any standards enforcing tool like `checkc` or `checkada`. Other methods of taking the users choice could also have been implemented.

A user places his choices about what constraints are part of the coding standard in a file called the *constraint selection file*. This file is the input to `makestd`. It has a special format. A menu-based program `makeinput` can be used to create the constraint selection file or the user can directly type out the constraint selection file and follow its format rules

Using `makeinput` is preferable because it saves time and leaves no scope for error.

3.1.2 Coding standard header files

For each supported language, there is a header file having a structure definition with a field corresponding to each supported constraint. A function definition is also defined to set each field of the structure to a default value. By default all constraint checks are turned off.

3.1.3 Format of the constraint selection file

All constraint selection files must have the line

```
LANGUAGE = <language_name>
```

at the very beginning to tell `makestd` what language is the input language for the standards enforcement tool. The `<language_name>` can be `C`, `Ada` or whatever else is supported.

Each constraint selection must be specified on a line by itself. Every constraint has been given a symbolic name and it can be specified in the constraint selection file as

```
<constraint_name> = yes|no|value
```

Giving *yes* turns on the check and giving *no* turns off that check. A numeric value has to be given for certain constraints that specify limits.

At the end of the constraint selection file, the word

END

must appear on a line by itself. Blank lines are ignored. Comments are not allowed, but this can be easily implemented. This format preserves the readability of the constraint selection file.

The lists of constraints supported for `checkc` and `checkada` are given in Appendices A and B. Mappings between the actual meanings of constraints and their symbolic names are given in the user-manuals for `checkc` and `checkada`. An example showing a sample input file to `makestd` is given below.

```
LANGUAGE = c
NOT_ASSIGN_CONDTTEST = yes
NO_PTRARITH_INEXPR = yes
NO_PTRARITH_INCONDTTEST = yes
NOT_ALLOW_CHARINDEX = yes
NO_FLDB_INCONDEXP = yes
SET_EXTNAMESIGLENGTH = 5
NOT_REUSE_OSCOPE_NAME= yes
SET_INTNAMESIGLENGTH = 5
NO_EXPR_SIDEFFS = yes
CHK_MISSING_ELSE = yes
CHK_DEEP_BREAKS = yes
CHK_BODYLESS_STMT = yes
CHK_BODY_NOT_BLOCK = yes
SET_MAXDEPTH_NESTING = 3
ENFORCE_ALL_NONEXTERN_STATIC = yes
SET_MAX_INCLUDE_DEPTH = 3
CHK_PRAGMA_INDENTED = yes
CHK_HEADER_OVERLOAD = yes
SET_MAXFIELD_STRUCT = 6
SET_MAXMEM_ENUM = 6
```

```

CHK_LOSSY_ASSIGN = yes
NO_SGN_UNSGN_ASSIGN = yes
SET_MAXLENGTH_STRLITERAL = 20
FLAG_SENSITIVE_LOOPS = yes
CHK_MULTIPLE_INCLUDE = yes
CHK_SIMILAR_NAMES = yes
CHK_MISSING_BREAK = yes
NOT_ALLOW_ENUMINDEX = yes
CHK_STORAGE_USED_AFTER_FREE = yes
CHK_STORAGE_NOT_FREED = yes
NO_CONCATENATION_OP = yes
NO_TOKEN_SUBSTITUTION = yes
VARS_INIT_AT_DECL = yes
CHK_RETURNOF_PTRTO_LOCAL = yes
WARN_STDFUNC_OVERLOAD = yes
WARN_VAR_NOTINIT = yes
CHK_PATH_WITH_NORETURN = yes
END

```

3.1.4 Output from makestd

After scanning the constraint selection file, an output file is created with the first part consisting of the header file of the target language. This consists of the structure definition and the function to initialize all the fields to their default values.

The second part of the output file is created by `makestd` during the scanning of the constraint selection file. It is a C function `set_struct_<language_name>()` that sets each field of the coding-standard structure according to the values specified by the user in the constraint selection file. A field is set to 1 if the corresponding check is to be turned on.

Figure 2 gives the schematic of `makestd`. The program can be extended to read constraint selections for additional languages. Currently it reads constraint selection files for C and Ada and generates code that is linked with `checkc` and `checkada`.

3.1.5 Integration with the constraint enforcing system

To integrate the generated coding standard file with the rest of the system,

1. It is compiled to object form
2. It is linked with the rest of the system i.e. object files of tools like `checkc` or `checkada`

Prior to parsing and checking an input file for conformance to the coding-standard, tools like `checkc` and `checkada` call the function

```
set_struct_<language_name>()
```

which is generated by `makestd`. This function call sets the structure field values in the coding standard structure to the desired values. `checkc` calls the function `set_struct_c()` and `checkada` calls the function `set_struct_ada()`.

Figure 3 shows how the coding standard file produced by `makestd` is compiled and linked with the object files of any standards enforcing tool (like `checkc` or `checkada`) to produce the final executable.

3.1.6 Introducing new constraints

It may be desired to introduce new constraints for the languages that are supported (C and Ada at present). Introducing a new constraint involves the following steps:

1. Giving a descriptive symbolic name to the new constraint
2. Changing the coding standard header file for that language to add a new field (corresponding to the new constraint) to the structure definition and initializer function code

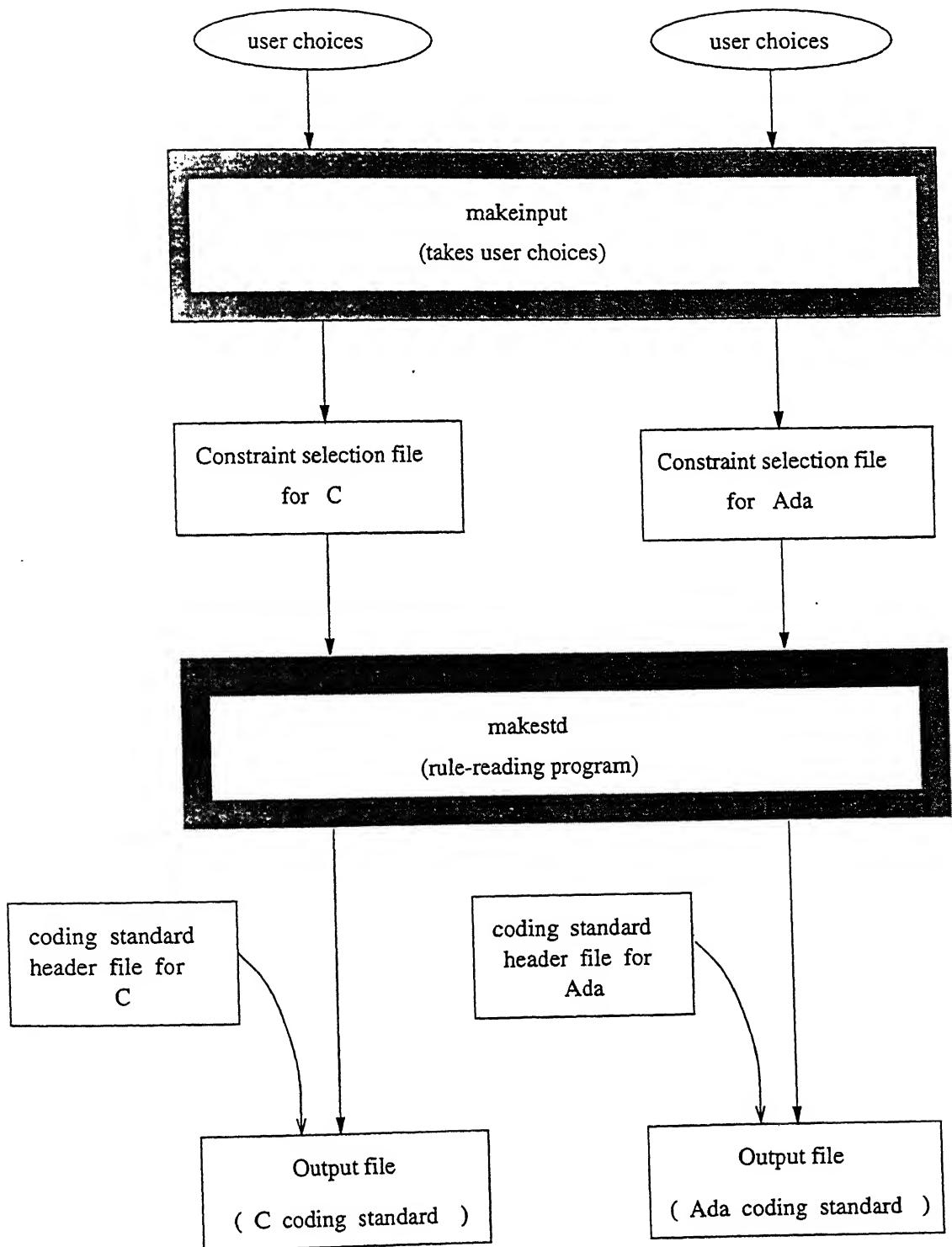


Figure 2: Action of the constraint specification process

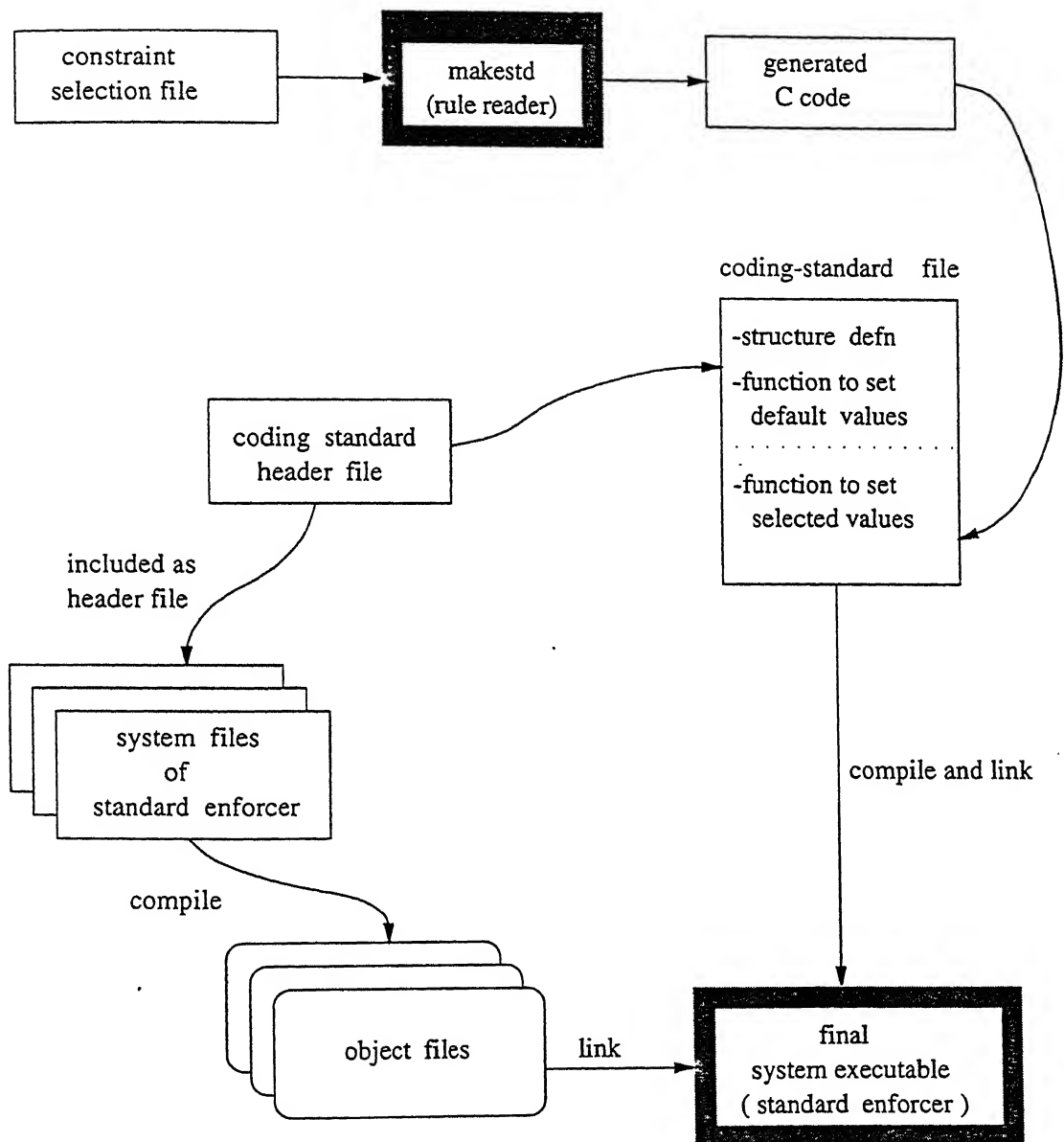


Figure 3: Schematic of constraint specification and system integration

3. Modifying `makestd` to recognize the new rule pattern and generate code to set the appropriate field in the coding-standard structure. This modification is an append operation and does not affect existing code in the program
4. Adding the code to enforce the new constraint at the appropriate places in the constraint enforcing tool (like `checkc` or `checkada`)

3.2 Enforcement of coding standards

3.2.1 Technique for enforcing constraints

Among the syntax constraints, style guidelines like those for horizontal spacing, indentation, alignment, pagination, line length, capitalization and naming conventions have not been dealt with. These are in the realm of **Style Checkers** and **Program Beautifiers**.

Syntax constraints are enforced either in the lexical analyzer, the parser or during parse-tree traversal. Usually no other information is needed apart from the token stream to enforce these constraints.

Semantic constraints are enforced in the parser during parsing or later during parse-tree traversal. It requires the type information about identifiers from the symbol table. So routines to build and maintain the symbol table as well as to insert and delete symbols from the symbol table have to be written. Type information on identifiers has to be inserted into the symbol table during parsing.

Enforcing dataflow constraints needs additional information to be put into the symbol table and other data structures. This is related to the use of variables in certain contexts. Implementing this type of constraint is the most difficult among the three types.

During parsing, the actions for each production are used to build the parse-tree from the bottom up. At each action, the roots of the sub-trees for the terminals and

non-terminals on the right-hand-side of the corresponding production are spliced together and become the immediate children of the subtree which is the value returned by the action. Each time a subtree is created by splicing together some other subtrees, a unique number is also stored in the node to identify the production used. So as many unique numbers are used as there are productions in the grammar. All the tokens returned by the lexical analyzer are leaf-nodes in the parse-tree.

Example of parse-tree construction during parsing:

```
exit_stmt : EXIT  name_opt  when_opt  SEMICOLON
{
    treenode *tmpnode; /* pointer to a node */

    /* allocate memory and store a unique number in
     * a 'type' field to represent use of this
     * particular production
     */
    tmpnode = make_node(1000);

    /* splice together the children */
    tmpnode->ptr_1 = $1;
    tmpnode->ptr_2 = $2;
    tmpnode->ptr_3 = $3;
    tmpnode->ptr_4 = $4;

    /* return the new node */
    $$ = tmpnode;
}
```

At the end of parsing, when the start symbol is reached, the parse-tree is traversed depth-first and from left-to-right. The parsed source code can be unparsed by printing out the appropriate lexemes for the tokens in the leaf nodes.

Almost all constraints are enforced during this depth-first left-to-right traversal of the parse-tree. It is much easier to enforce constraints during a top-down traversal after parsing rather than during bottom-up parsing. This is because in the top-down traversal, the higher level contexts like conditional statement, loop statement, function body and so forth can be known at any point of time. This information is useful in enforcing a large number of constraints that deal with using or not using language features in certain contexts.

3.2.2 Examples of constraint enforcement

So for example, if we want to enforce a semantic constraint *real numbers should not be used in conditional expressions*, we set a flag when we reach a node which represents the use of the production for conditional expression. This is known from the unique number stored in one of the fields in the node. Now, when we visit the children of this node and encounter a node for a variable use and the flag is set, we get the type of the variable from its symbol table entry and if it happens to be a real number, a warning message is generated. The flag set earlier can be reset when all the children are visited.

In some situations, the higher level context within which the constraint is to be enforced may be nested. So instead of setting and resetting the flag, it can be incremented and decremented by one before and after visiting the children of each node for such higher level contexts.

Another example can be to enforce the dataflow constraint *all pointer variables should be explicitly freed before their scope is exited*. This can be implemented in the following way. Whenever we get a declaration for a pointer variable and create its symbol table entry, another field is also marked in the entry indicating that it is not yet freed. Now whenever we come to the node for a function call, we check the subtree below it to see if it is a call to the function that frees memory. If so, we find what variable is being freed by checking the subtree again to get the argument of the memory freeing function. If the symbol table entry of that variable indicates

that it is not yet freed, that entry is set to indicate that the variable has been freed. Also at the nodes which indicate scope exit—like the ends of function bodies, ends of blocks and so forth, the symbol table is scanned for all variables about to go out of scope and if any are pointer variables and have the not-freed indicator still set, warning messages are printed about these variables. This approach fails for certain cases like when a pointer is freed within a called function rather than the one in which it is declared.

3.2.3 Generation of warnings

When a constraint that is part of the set coding standard is violated, the enforcement tool has to generate a warning and warn the user about the violation.

In **checkc**, the input C source code to be checked is printed out again, annotated with C comments at places where the set constraints have been violated. These comments describe the violations. Some warnings are also produced after printing the commented file. These warnings have a line number and filename to help locate the causes of the warnings. An example input program to **checkc** and the output produced is given in the user manual for **checkc**.

In **checkada**, first the input Ada file is printed with line numbers shown at the start of each line. Then the warnings are printed. Each warning has a line number to help locate the constraint violation. An example input program to **checkada** and output produced is given in the user manual for **checkada**.

Figure 4 shows the action of tools like **checkc** and **checkada** in terms of how they act on the input files and where warnings are produced for violations of the coding standard.

The user has to read the warnings generated for constraint violations and modify the code to remove the violations. The code can be modified and submitted repeatedly to the enforcer tool until no warnings are produced.

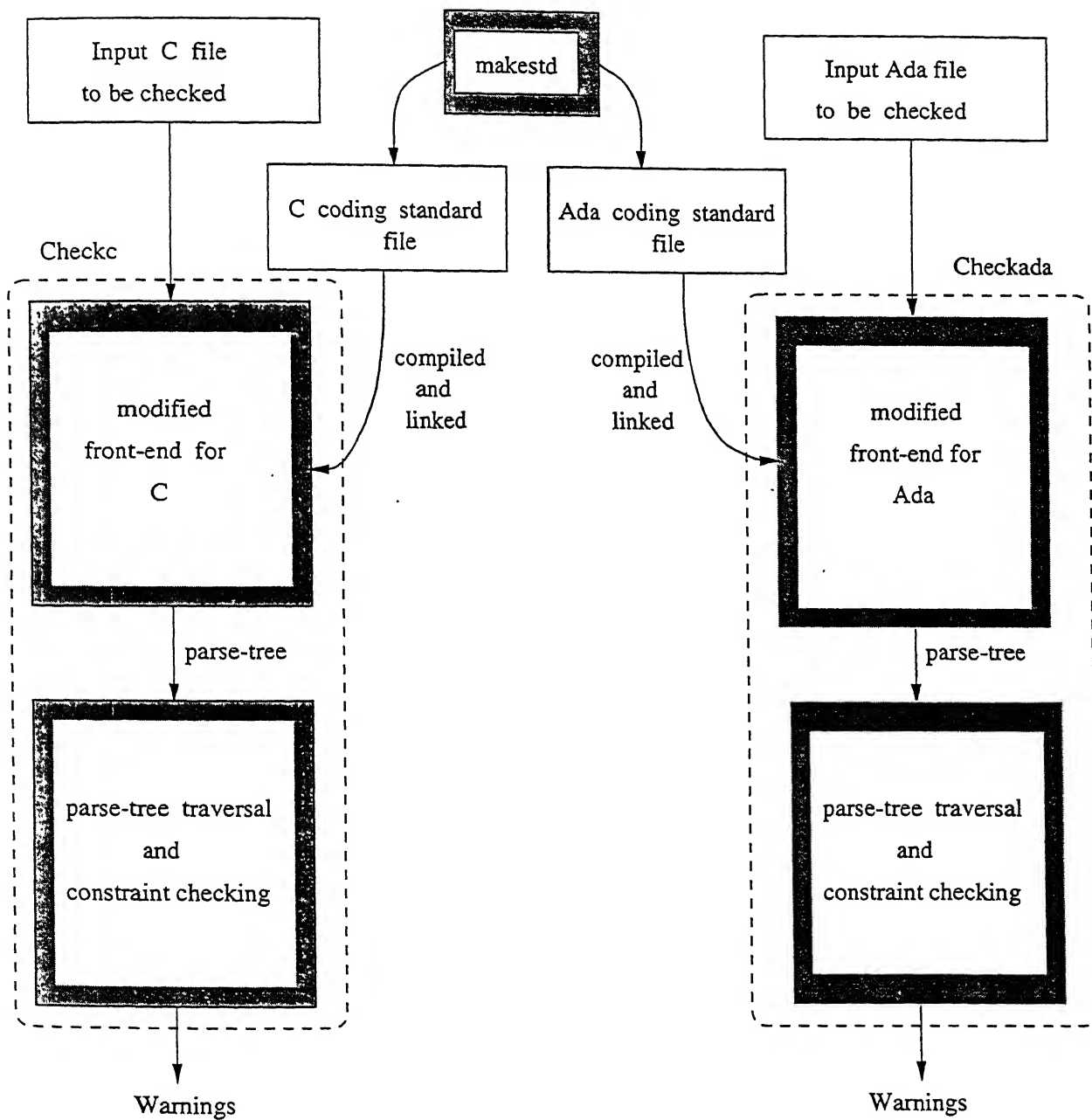


Figure 4: Action of the constraint enforcing tools

3.2.4 Automatic restructuring to remove violations

In certain cases of syntax-constraint violations, it is possible to remove these violations automatically during the unparsing of the parse tree by printing additional lexemes. These cases are however, very small in number.

For example, if the constraint that *all if-then-else statements should have else parts* is set for enforcement. The following code segment would violate this constraint.

```
if ( varx == vary ) {  
    printf("varx and vary are equal \n");  
}
```

During traversal of the parse-tree nodes, the lack of an *else* part of the *if-then-else* statement is identified and the code printed for the above fragment is

```
if ( varx == vary ) {  
    printf("varx and vary are equal \n");  
}  
else {  
}
```

Chapter 4

Generation of coding standard enforcers

4.1 Benefits of a Generator Tool

During the implementation of `checkc` and `checkada` it was found that manually constructing constraint enforcers is a tedious and error prone process. Much time and effort would be saved if a way could be found to generate a constraint enforcing tool automatically from user specifications of the language grammar and what checks to enforce.

This has the added incentive that even if some new, currently unheard of constraint is to be introduced later for a supported language, the required code to enforce the constraint would still be generated by the tool. The tool would also generate enforcers for new languages that are used in the future. To achieve the goal of a standards enforcer Generator Tool, a formal way of specifying what checks to enforce has to be found. The generator tool would read the specifications, analyze the grammar to find out where to generate code to enforce constraints and generate appropriate code.

However, no way could be found to automatically understand specifications of checks to enforce and to generate code for doing the checks.

4.2 createparse—a tool for partial automation

A way was found to achieve part of the goal of building a standards enforcer Generator Tool. A tool `createparse` was developed which can read the user supplied specification of any language grammar and generate a substantial part of the front-end for that language. It can generate a skeleton lexical analyzer (with blank actions), a full bison input file with action parts to construct the parse-tree and a function to traverse the parse-tree and print the lexemes for all the tokens in the leaf nodes.

The front-end that is generated by `createparse` for the specified language has to be manually modified to make it a coding standard enforcing tool.

`createparse` accepts a specification of any language grammar in a file named *language specification file*.

4.2.1 Format of the language specification file

The format of the language specification file that is input to `createparse` is given below

```
%{{tokens
    <token_1>    <corresponding_lexeme>
    <token_2>    <corresponding_lexeme>
    .
    .
    .
    <token_n>    <corresponding_lexeme>
%}}

%{{lex_subs
    <short_form_1>    <corresponding_regular_expression>
    <short_form_2>    <corresponding_regular_expression>
```

```

.
.
.
<short_form_n>    <corresponding_regular_expression>
%}}

%{{lang_keywords_tokens
    <token_1>      <corresponding_keyword>
    <token_2>      <corresponding_keyword>
    .
    .
    .
    <token_n>      <corresponding_keyword>
%}}

%{{max_numright_children <number>
%}}

%{{grammar
    <bison grammar specification here>
%}}

```

The *tokens* section has the lexical tokens and their associated lexemes. The lexeme part may also be a regular expression.

The *lex_subs* section has the aliases of regular expressions (that are very often used in lexical analyzer specifications to save typing) and their associated regular expressions.

The *lang_keywords_tokens* section lists the keywords of the language and their corresponding tokens returned to the parser. This is used to create a function in the generated lexer that scans a list of the keywords and matches them with it's

argument. The function returns tokens corresponding to the matched keywords to the parser.

The *max_numright_children* section gives the maximum number of elements on the right-hand-side of a production among all the productions in the grammar. This number determines the maximum number of child pointers any node in the parse-tree may have and is necessary to generate the correct structure definition for the parse-tree nodes.

The *grammar* section lists the LALR(1) context-free grammar of the language (without any action parts).

The LALR(1) context-free grammar for the language that is specified in the *grammar* section is parsed and the bison input file produced by a separate program named *genbison* called by *createparse*.

4.2.2 Building the complete front-end

Before compiling the generated front-end, first the complete lexer has to be created from the generated skeleton by adding the actions. Then the generated front-end can be compiled by using the makefile named *auto_makefile* that is also generated. Actions in the generated lexer specification cannot be automatically created because they are different for different lexical patterns. In some, a token is unambiguously identified and returned to the parser. In others, like the pattern that matches all keywords and identifiers, the matched lexeme may have to be matched against entries in a list before knowing which token to return.

The idea for making *createparse* came from [19]. The format of the language specification file that is input to *createparse* is also somewhat similar to that described in [19], as is the basic scheme of operation.

Figure 5 shows the files that are generated by *createparse* and *genbison*. Figure 6 gives a more abstract view of the whole process.

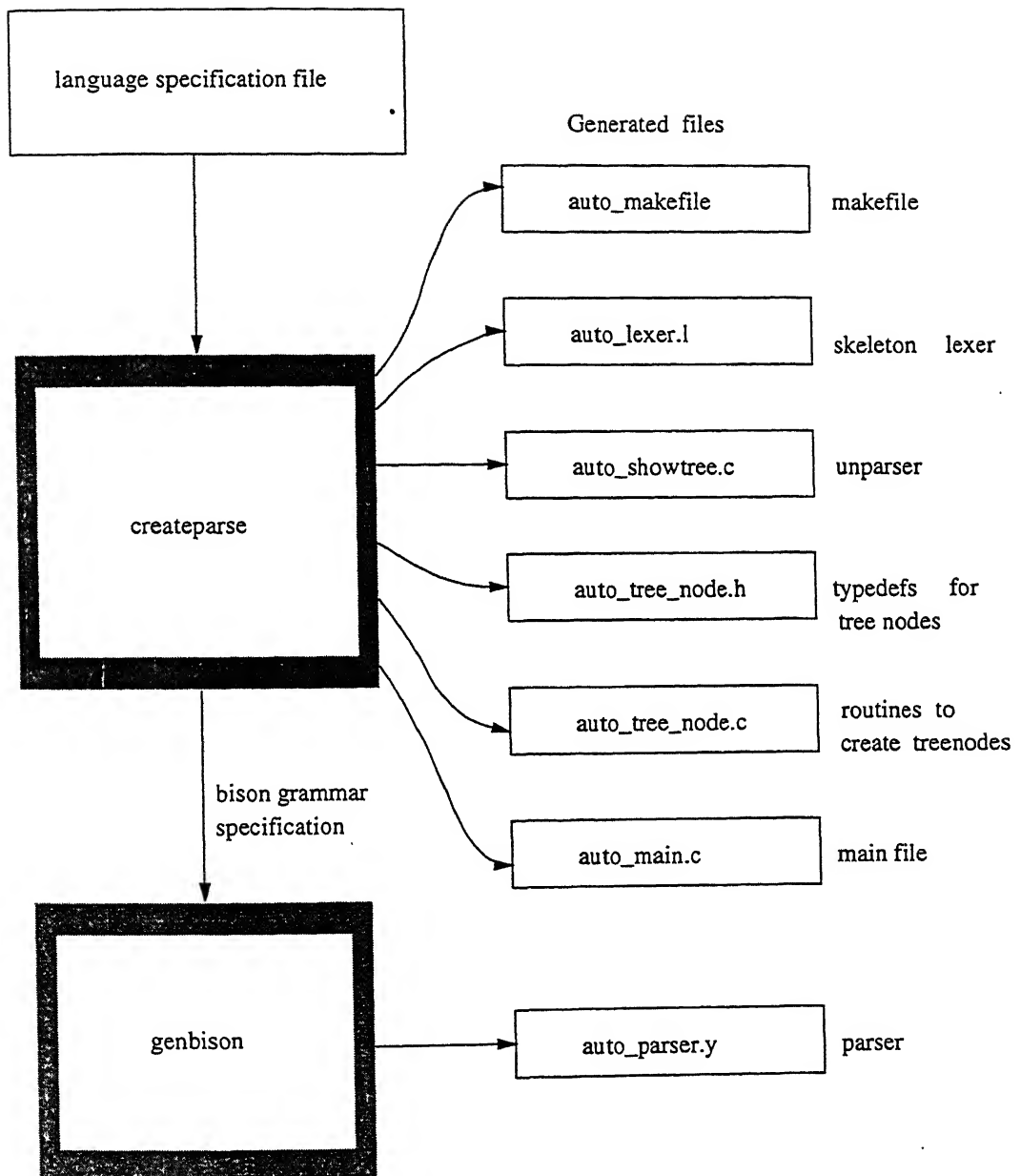


Figure 5: Action of createparse and genbison

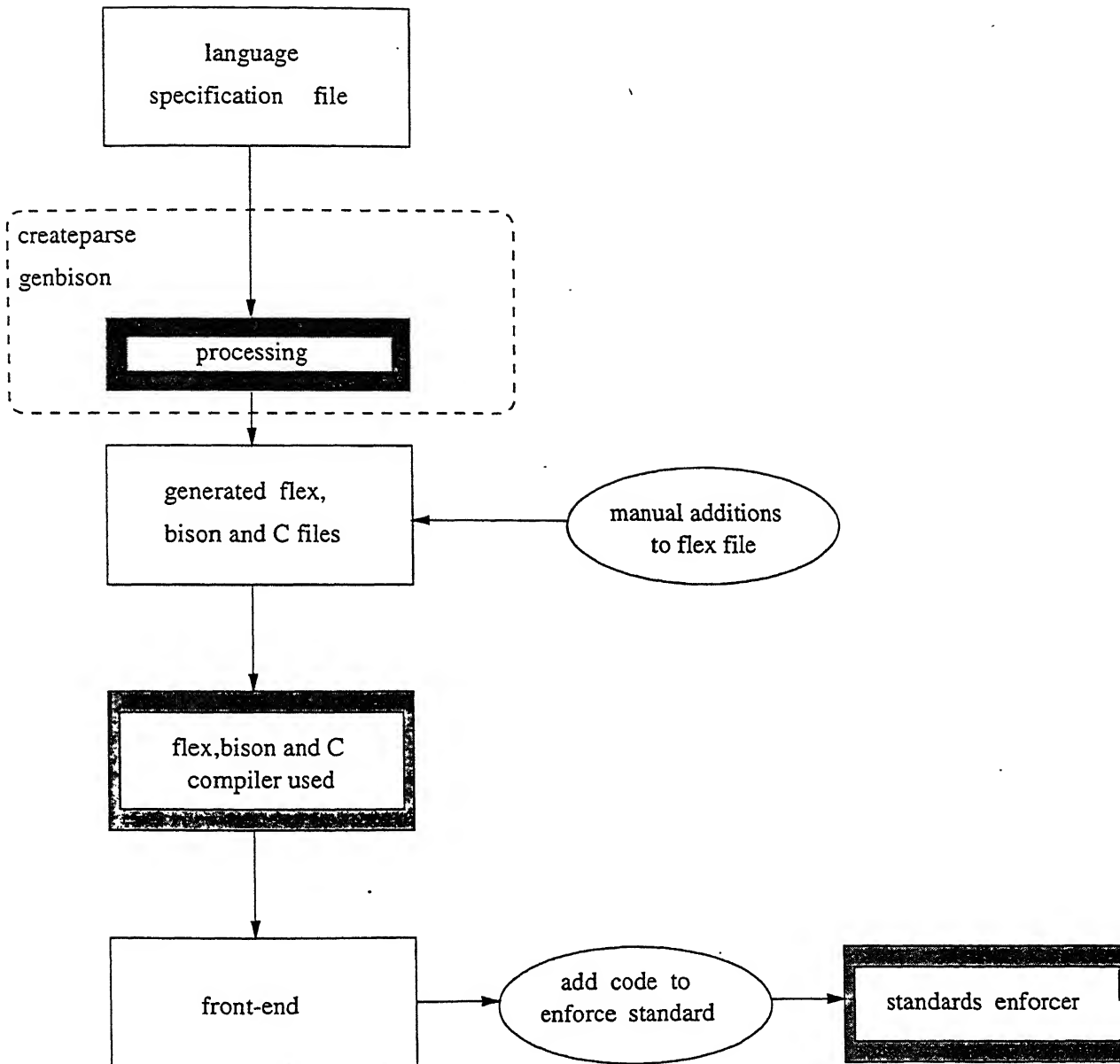


Figure 6: Generation of the skeleton front-end

Chapter 5

Implementation details of the Tools

5.1 A note on Flex and Bison

Flex is a tool for generating scanners: programs which recognize lexical patterns in text. Flex reads the given input files (or its standard input if no filenames are given) for a description of the scanner to generate. The description is in the form of pairs of regular expressions and C code, called *rules*. Flex generates an output C source file `lex.yy.c`, which defines a routine `yylex`. When the executable runs, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code. It has some differences with the older tool `lex`. Generally the scanners generated by flex are faster than the ones generated by `lex`.

Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse the grammar. Bison is backward compatible with yacc—all properly written yacc grammars ought to work with bison with no change.

The software created during this thesis do not work with `lex` and `yacc`. Flex and bison can be obtained from any GNU ftp site.

5.2 Languages and development tools used

`makestd` was made using `flex`. It reads a user supplied file that specifies what checks are to be enforced. It produces a C structure definition which encodes the user choices in it's fields. This structure definition is then used by the standards enforcing tool for the selected language to find out which checks to enforce.

A freeware package named `ctree`[24] was modified to create the constraint enforcer for C named `checkc`. `checkc` consists of a lexer, parser, symbol-table routines, code to build and traverse the parse-tree and code to enforce the supported constraints.

The Ada constraint enforcer named `checkada` was developed from scratch. The grammar for Ada95 was used as the starting point. Its components are the same as in `checkc`.

`flex` and `bison` were used to construct the front-ends in both `checkc` and `check-ada`.

The Generator Tool, `createparse` uses `flex` and `bison` to read the language specification file that is input to it.

All the software created are in the C language. The default `cc` C compiler was used to compile the software. `make` was used to make the executables. Development was done on a DEC-2000 system running DEC/OSF-1.

5.3 Statistics about the software developed

Some size and effort metrics about the software are given below:

software	total size	prior code	newly developed	person months
makestd,makeinput	1618	0	1618	0.50
checkc	8251	4957	3294	2.00
checkada	11607	1378	10229	1.50
createparse	1483	0	1483	0.50

As can be seen, `checkc` took a disproportionately large slice of the time spent on coding when compared to its Lines of Code. This happened because it was developed prior to `checkada` and all the techniques used later were discovered during the course of its development. There were many mistakes made and wrong decisions taken which had to be corrected later. Once the techniques were firmed up, `checkada` was developed rapidly.

5.4 Development of `checkc`—the C constraint enforcer

The starting point for `checkc` was `ctree`, a simple package which parses ANSI C programs and builds a parse-tree. There were a large number of bugs in `ctree`, both in the ANSI C grammar and in the code. These were rooted out eventually. Then additional code was added to do 37 constraint checks. A listing of all constraint checks done by `checkc` is given in Appendix A. It was found that rather than base a constraint enforcer on an existing package and having to modify it and also accept any design flaws and correct existing bugs, it would be better to start from scratch. i.e from the grammar itself.

The 37 constraints enforced by `checkc` were gathered from [7], [8], [9], [10], [11], [13] and [14].

Testing of this tool was first done with small programs with violations deliberately introduced in order to see if the violations would be detected. Later larger C files which are part of `checkc` itself were used as test input. The largest file input was a 2500 line C file which is the main file of `checkc` itself.

5.5 Development of `checkada`—the Ada constraint enforcer

Given the experience with `checkc`, `checkada` was developed from scratch starting with the Ada95 grammar. It was found that this approach was much faster because the developer could support the ultimate goals from the start itself. Also the experience gained from `checkc` led to many wrong decisions being avoided. A listing of the 47 constraints supported by `checkada` is given in Appendix B.

All the 47 constraints enforced by `checkada` were taken from [3].

Testing of `checkada` was done with a collection of 35 small Ada programs which came along with the Ada95 grammar. No testing could be done with a very large Ada program as input, because it was not available.

5.6 Development of `createparse`—the Generator Tool

`createparse` was developed after the completion of `checkc` and `checkada`. Language specification files were created to describe C and Ada and front-ends were successfully generated and tested for both languages.

Using `createparse` as the first step in creating a standards enforcing tool frees the user from most of the initial work of building a front-end for the language. This saves a few days of development time. It lets him focus on the actual work needed to enforce the constraints.

Chapter 6

Conclusion and further work

The techniques developed during the thesis for enforcing coding standards were general enough to be used successfully for both the standards enforcing tools, `checkc` and `checkada`. Enforcers can be made for other languages using similar techniques. The `createparse` tool will help in such efforts and automate most of the front-end construction. Moreover, it will also ensure a degree of similarity between all such tools.

Of the original desired features, the two standards enforcing tools meet all of them to a substantial degree.

- Both are flexible and allow the user to set coding standards from a fairly large collection of constraints
- The method of specifying which constraints to enforce is simple to use
- Documentation is quite thorough. A full set of user manuals is provided in HTML format as well as design documents for all the software
- Both the tools are easy to use
- Feedback provided to the user is sufficiently informative

The final goal of implementing a Generator Tool to automatically create enforcing tools for different languages could not be realized. The partial automator tool `createparse` only generates a skeleton front-end for any language. This front-end has to be manually enhanced to a standards enforcement tool.

6.1 Comparisons with existing tools

As far as functionality is concerned, `checkc` offers more fine-grained degree of control over the coding standard than `lint`. `LCLint`[7], at the expense of the user annotating the code with special comments, does several hundred constraint checks compared to only 37 by `checkc`.

It is difficult to compare `checkada` because the commercial tools like `Ada-Assured`[23] are not freely available. The brochure for `Ada-Assured` states that it enforces all the several hundred coding guidelines in [3]. `checkada` only enforces 47 guidelines from [3].

6.2 Limitations

1. A rigorous testing process is necessary to ensure that software produced is of high quality. This could not be done for the software developed due to lack of time and manpower.
2. The method of specifying constraint selections to `makestd` is rudimentary at the moment. This can be made more sophisticated by developing a more structured method for specifying constraint selections.
3. Because Ada95 has a very large grammar with over 600 productions, and the fact that memory is allocated and tree-nodes are built up at every use of every production during parsing, the parser generated by `bison` and which forms part of `checkada` is slow in operation. This slowness will be quite noticeable when large Ada programs are input for checking.

4. Both `checkc` and `checkada` generate spurious warnings about non-existent violations in some situations. They also fail to warn about constraint violations in certain complex cases. More extensive testing is necessary to identify all such problems.
5. There is no method to check that the language specification file input by the user to `createparse` is correct. An incorrect specification file will result in an incorrect front-end being generated.

6.3 Further work

Apart from fixing bugs and further testing, further work to enhance the existing system would be:

1. Enforce more constraints for C and Ada
2. Build constraint enforcers for other important languages like Pascal and C++
3. Create a graphical user interface to handle interaction with any coding standard enforcer like `checkc` or `checkada`

A new avenue would be to study the feasibility of generating constraint enforcers automatically from user supplied specifications of constraints and the language grammar. Syntax-based editor generators like the **Synthesizer Generator** offer promise.

Appendix A

List of constraints enforced by checkc

All the constraints supported for C are given along with the rationale for enforcing them.

- **Syntax constraints**

Check if statements with missing else parts although it may be perfectly reasonable in certain contexts, an if-else chain with no final else may indicate missing logic or forgetting to check some error condition

Check for breaks in multiply nested loops breaks from deeply nested loops make it difficult for the reader to find the point where execution jumps to

Check for statement with no body an empty if, while or for statement often indicates a potential bug

Check if the body of certain statements is not block readability is improved if if, while, do-while and for statements have bodies that are blocks

Set the maximum statement nesting depth readability is improved

Set the maximum nesting depth for include files understandability improved. ANSI C has a limit of 8

- Check that all `#pragmas` are indented makes the program compilable by compilers that use older preprocessors
- Check if header names clash with common system headers potential confusion for the maintainer is avoided
- Check `printf` and `scanf` for correct use this checks that the number of format specifiers and the number of actual arguments match for `printf` and `scanf` . takes care of a common source of error in C programs
- Set the maximum number of fields that a structure may have some compilers have small limits on this number. this constraint can be used to take care of this concern. ANSI C has a limit of 127
- Set the maximum number of fields that an enum may have some compilers have a low limit on this number
- Set the maximum length of string literals readability improved
- Warn about multiple includes of the same file prevents compile-time error
- Check for similar looking names removes a source of potential confusion for developer and maintainer
- Check for missing break in case statements catches a common error in C
- Warn about use of token concatenation in macros ANSI C standard defines the operator `##` for token concatenation
- Warn about use of token substitution in macros some preprocessors perform token substitution within quotes while others do not. Therefore, this is intrinsically non-portable. ANSI C disallows it
- Set the significant length for internal names some compilers have limits i.e all internal names should be significant within some number of characters
- Set the significant length for external names some compilers have limits i.e all external names should be significant within some number of characters

Warn if function names clash with system function names removes a source of confusion for developer and maintainer

• Semantic constraints

Conditional test should not be assignment this usually indicates a careless typing error by the developer

Expressions should not have pointer arithmetic this removes a leading cause of errors in C

Conditional test should not have pointer arithmetic avoids errors and makes code more readable

Characters not allowed to index arrays characters have a limited range and would not be able to index large arrays

No float or double variable allowed in conditional expressions errors can be introduced in using `==` and `!=` operators to compare floating point types. Expressions such as $(float_expr_1 == float_expr_2)$ will seldom be satisfied due to rounding errors. For example, if we try to evaluate $10^{50} + 812 - 10^{50} + 10^{55} + 511 - 10^{55} = 812 + 511 = 1323$, most computers will produce 0 regardless of whether one uses float or double

Warn about same name reused in different scopes this removes a potential cause of errors. Also the masked off name(s) cannot be used directly in the inner scope unless we have another variable pointing to them

Disallow expressions with side-effects expressions having side-effects always complicate the understanding of the code and makes maintainance difficult

All non-extern global variables should be static this will restrict the scope of variables to the file where they are declared and prevent name clashes which will cause linker errors

Check for loss-making assignments often C will silently do a truncation and do the assignment. This can lead to a loss of information and change the behaviour of the program

Check for assignments between signed and unsigned variables this warns about assignments between signed and unsigned variables. An unsigned type cannot represent all signed values correctly

Flag loops whose number of iterations depend on variables this helps the developer to review a potential trouble-spot in the code after typing it out

Disallow enum variables to index arrays the enum variable may not be able to get the value necessary to index properly

warn if function returns a pointer to a local variable local variable will be deallocated on function exit. So this catches an unsafe programming practice

• Dataflow constraints

Check for dynamic storage used after freeing compiler does not check this. The state of storage is undefined and will probably lead to a runtime error or even worse, silently change the program behaviour

Check for dynamic storage not freed prior to scope exit it is always good to free explicitly rather than relying upon anything else. Sometimes even when out of scope, a variable's storage may be accessible due to implementation errors in the compiler

Check for paths in functions having no return statements it is important to check this in case the function returns a non-void type but the compiler does not check this.

variable not initialized on atleast one path is used this prevents a potential runtime error

Appendix B

List of constraints enforced by checkada

All the constraints supported for Ada are given along with the rationale for enforcing them.

- **Syntax constraints**

Associate names with loops When we associate names with loops, we must also include that name with the associated end for that loop. This helps readers find the end of any nested loop easily. This is useful if loops are broken over a screen or page boundary

Associate names with blocks The benefit is the same as with associating names with loops

Use the loop name on all exit statements from nested loops It can be difficult to determine which loop is being exited in the case of a loop nest otherwise.

Include simple name at end of package specification and body Readability is improved

Include simple name at end of task specification and body Readability is improved

Include designator at end of subprogram body Readability is improved

Show mode indication of all procedure parameters Aids the reader.

Default mode is in. Anyone reviewing the code later will be more confident if in is used explicitly rather than implicitly

Use range constraints on all subtypes This gives the benefit of the compiler's constraint checking.

Avoid anonymous types Although Ada allows anonymous types, they have limited usefulness and complicate program modification. For example, a variable of anonymous type can never be used as an actual parameter because the type of the formal parameter will be different

Always use an others choice in case statements Absence of others choice when user had not enumerated all possibilities leaves it open to raising an exception should the omitted possibility arise. The statement that *it can never happen* is an unacceptable programming practice

Avoid exit statements in while and for loops It is a bad idea to use an exit statement from within a for or while loop because it is misleading to the reader after having apparently described the complete set of loop conditions at the top of the loop

Avoid goto statements A goto statement is an unstructured change in the control flow. Worse, the label does not require an indicator of where the corresponding goto statement(s) are. This makes code unreadable and makes its correct execution suspect

Avoid abort statements When an abort is executed, there is no way to know what the targeted task was doing beforehand. Data for which the target task is responsible may be left in an inconsistent state. It is possible for a task to mistakenly execute abort statements targeting tasks it does not intend, including itself, due to aliases or the tree of task dependency. Further, the abort statement is not useful for dealing with tasks that are no longer accessible

Use a parameterless procedure as the main program Ada places few requirements upon the main subprogram. Assuming the simplest case increases portability. The user should avoid using any implementation features associated with the main subprogram

Avoid machine code inserts This is an implementation dependent feature. Machine code insertion can be achieved by special procedures whose declarative parts contain at most use clauses and pragmas and whose only statements are special statements called machine code statements. No exception handlers are allowed in such procedures. The facility is tedious and difficult to use, so its indiscriminate use is not encouraged. A predefined package `MACHINE_CODE` is assumed available. Implementations are not required to provide the `MACHINE_CODE` package. Differences in lower level details like register-conventions would hinder portability

Avoid interfacing Ada to other languages The problems with employing pragma `INTERFACE` are complex. The problems include pragma syntax differences, conventions for linking/binding Ada to other languages and mapping Ada variables to foreign language variables, among others

Do not overload the typenames in the package `STANDARD` Removes potential confusion later for the maintainer

Do not overload enumeration literals

eg. of overloading

```
type myA is (red, green, blue);  
type myB is (pink, red, yellow);
```

Understanding the code is easier

set maximum limit on number of subprogram parameters Understandability and readability is improved

Avoid nested package specifications Nesting of package specifications should be done only when grouping operations, hiding common implementation

details or presenting different views of the same abstraction

Avoid nested tasks in package specifications Hiding a task specification in a package body and exporting (via subprograms) only required entries reduces the amount of extraneous information in the package specification.

Put specification and body of all packages in different files Helps the maintainer to understand code faster

Use type qualifiers rather than type casting type casting is a dangerous practice in most circumstances

Avoid using anonymous tasks The use of tasks of anonymous type would avoid a proliferation of task types that were only used once, and the practice does communicate to maintainers that there are no other task objects of this type. However, a program originally written using named tasks that must be enhanced to contain several tasks of the same type will require more work to upgrade than the same program written with task types

Avoid conditional entry calls Using this construct poses the threat of race conditions. This construct is very much implementation dependent. For conditional entry calls and selective waits with else parts, Ada does not define *immediately*. For timed entry calls and selective waits with delay alternatives, implementors may have ideas of time that differ from each other and from the users

Avoid selective waits with else parts

Avoid timed entry

Avoid selective wait with delay alternatives

Avoid explicitly raising predefined exceptions User defined exception names can be made more descriptive in a particular situation than the predefined exception names. In addition, there already are too many situations that result in predefined exceptions. Creating additional causes for predefined exceptions increases the difficulty of determining the source of an

- exception. An exception name should have one and not several meanings
- Avoid variable declarations in package specifications Variables could have get() and set() functions in the specifications and be declared in the package body
- Use named association in subprogram calls Calls to subprograms with many formal parameters can be difficult to understand without referring to the subprogram or entry code. Named parameter association makes these more readable. When the formal parameters have been named appropriately, it is easy to determine exactly what purpose the subprogram serves without looking at its code. This reduces the need for named constants that exist solely to make calls more readable. It also allows variables used as actual parameters to be given names indicating what they are, without regard to why they are being passed in a call. An actual parameter that is an expression rather than a variable cannot be named otherwise
- Avoid using **UNCHECKED_CONVERSION** An unchecked conversion is a bit-by-bit copy without regard to the meanings attached to those bits and bit positions by either the source or the destination type. The source bit pattern can easily be meaningless
- Avoid using **UNCHECKED_DEALLOCATION** The unchecked storage deallocation mechanism is one method for over-riding the default time at which allocated storage is reclaimed. The earliest default time is when an object is no longer accessible eg. when control leaves the scope where an access type is declared (the exact point after this time is implementation dependent). Any unchecked deallocation of storage performed prior to this may result in an erroneous program if an attempt is made to access the object
- Initialize all access variables and components within a record
- Initialize all variables at declaration Makes understanding the code easier

• Semantic constraints

Deallocate all access variables explicitly This documents the decision to abandon an object

Do not use real operands if the comparison operator is `=` `<=` and `>=` should be used in relational expressions with real operands instead of `=`. Fixed and floating point values, even if derived from similar expressions, may not be exactly equal. The imprecise, finite representations of real numbers in hardware always have round-off errors so that any variation in the construction path or history of two reals has the potential for resulting in different numbers even when the paths or histories are mathematically equivalent

Do not use ranges of enumeration literals in case statements

eg.

```
type Color is (red, green, blue, purple);
car_color : Color := red;
.....
case car_color is
when red..blue => ...
when purple => ...
end case;
```

now consider a change of the type:

```
type Color is (red, yellow, green, blue, purple);
```

This change may have an unnoticed and undesired effect in the case statement. If the choices had been enumerated explicitly, as *when red | green | blue =>* instead of *when red..blue =>*, then the case statement would not have compiled. This would have forced the maintainer to make a conscious decision about what to do in the case of *yellow*

Do not reference non-local variables in a guard Prevents certain race conditions

Do not allow enumeration variables to index arrays Enumeration variables may not be able to store whole range of possible index values

• Dataflow constraints

Initialize all variables before use Ada does not define an initial default value for objects of any type other than access types. Using the value of an object before it has been assigned a value will raise an exception or result in erroneous program execution

Catch creation of unreferenced memory In the following three lines, if there is no intervening assignment of P1's value to any other pointer, the object created on the first line is no longer accessible after the third line

```
P1 := new OBJECT;  
.....  
P1 := P2;
```

Avoid dangling references The creation of a dangling reference is shown below.

```
P1 := new OBJECT;  
P2 := P1;  
UNCHECKED_OBJECT_DEALLOCATION( P2 );
```

now P1 refers to non-existent storage and is a dangling reference

Appendix C

User manuals

Location of the manual

This user manual is kept in HTML format in the `man` subdirectory of the distribution as `manual.html`

Hardware and Software requirements

Hardware Platform: no specific requirement. The software was developed on a DEC-2000 machine under DEC/OSF-1

Compilers: either `cc` or `gcc`

Tools: `make`, `flex` and `bison`

Contents of the distribution

The distribution after being untarred has five directories. They are described below along with their contents

archive: has ctree-0.02.tar.gz, the package on which checkc is based

code: has the subdirectories checkc, checkada, save, auto, makestd and cbeau. Their contents are described below

checkc: has the code of checkc

checkada: has the code of checkada

save: tar.z forms of all the tools in the distribution

auto: has the code of createparse and genbison

makestd: has the code of makeinput and makestd

cbeau: this is a modified version of checkc and is not to be considered part of the distribution

man: has the manuals of all the programs in HTML format

refs: has several directories that contain C and Ada coding standard guidelines as well as some general information files on compiler tools in HTML format

report: has the report, user-manual and design document in Latex form. The .lj forms of both the report-cum-manual and design documents are also kept there (report.lj and techdoc.lj)

Compiling the distribution

The code for the distribution is located in several sub-directories under the directory named code. The code directory has a makefile which takes care of making the executables for all the programs. Each individual program directory also has its own makefile.

Name

`makestd` creates a coding standard file from user supplied constraint selection file.

Synopsis

```
makestd [-o <outputfile>] < <filename>
```

`makestd` reads the user supplied constraint selection file and generates a function that sets the appropriate fields in the coding-standard structure. This function is appended to the declaration of the coding-standard structure for the language specified in the constraint selection file to create the output file. This file is linked with the constraint checker to be created.

Flags

`-o` This allows the user to specify an output filename. In the absence of this flag, the default output file is `std.output.c`

Description

Format of the constraint selection file

In case the choice is not to check the constraint, then nothing is done, otherwise output is generated such that the field for the constraint in the C structure representing the coding standard is set to the appropriate value. This value can be 1 or some other value depending on the nature of the constraint.

All input files *must* have the line

LANGUAGE = <language_name>

at the very beginning to help the makestd use the proper code within itself. The <language_name> can be C, Ada or whatever else is supported.

Each constraint should be specified on a line by itself. The end of the input file should be specified by

END

on a line by itself. This format is designed to preserve the clarity of the input file.

Input constraints supported by checkc

The following constraints (with meanings alongside) may be specified in the input file.

constraint name	meaning

NOT_ASSIGN_CONDTEST	conditional test should not be assignment
NO_PTRARITH_INEXPR	expressions should not have pointer arithmetic
NO_PTRARITH_INCONDTEST	conditional test should not have pointer arithmetic
NOT_ALLOW_CHARINDEX	characters not allowed to index arrays
NO_FLDB_INCONDEXP	no float or double variable allowed in conditional expressions
SET_EXTNAMESIGLENGTH	set the significant length for external names. all external names should be unique

	inside this length
NOT_REUSE_SCOPE_NAME	warn about same name reused in different scopes
SET_INTNAMESIGLENGTH	set the significant length for internal names.all internal names should be unique inside this length.
NO__EXPR_SIDEFFS	disallow expressions with side-effects
CHK_MISSING_ELSE	check 'if' statements with missing else part
CHK_DEEP_BREAKS	check for breaks in multiply nested loops
CHK_BODYLESS_STMT	check 'for' statement with no body
CHK_BODY_NOT_BLOCK	check if the body of certain statements is not block
SET_MAXDEPTH_NESTING	set the maximum statement nesting depth
ENFORCE_ALL_NONEXTERN_STATIC	all non-extern global variables should be static
SET_MAX_INCLUDE_DEPTH	set the maximum nesting depth for include files
CHK_PRAGMA_INDENTED	check that all #pragmas are indented so as to cause no problems on older preprocessors
CHK_HEADER_OVERLOAD	check if user supplied headers clash with the names of common system defined header

files

CHK_NUMPARAMS_PRSC	check that the number of format specifiers and the number of actual arguments match for printf,scanf
SET_MAXFIELD_STRUCT	set the maximum number of fields that a structure may have
SET_MAXMEM_ENUM	set the maximum number of fields that an enum may have
CHK_LOSSY_ASSIGN	check for loss-making assignments
NO_SGN_UNSGN_ASSIGN	check for assignments between signed and unsigned variables
SET_MAXLENGTH_STRLITERAL	set the maximum length of string literals
FLAG_SENSITIVE_LOOPS	flag loops whose number of iterations depend on variables for manual inspection
CHK_MULTIPLE_INCLUDE	warn about multiple includes of the same file
CHK_SIMILAR_NAMES	check for similar looking names
CHK_MISSING_BREAK	check for missing 'break' in 'case' statements
NOT_ALLOW_ENUMINDEX	disallow enum variables to index arrays
CHK_STORAGE_USED_AFTER_FREE	check for dynamic storage used after freeing

CHK_STORAGE_NOT_FREED	check for storage not freed prior to scope exit
NO_CONCATENATION_OP	warn about use of token concatenation in macros
NO_TOKEN_SUBSTITUTION	warn about use of token substitution in macros
VARS_INIT_AT_DECL	check that all scalar variables are initialized at declaration time itself
CHK_RETURNOF_PTRTO_LOCAL	warn if any function returns a pointer to a local variable(whose storage would be deallocated on return)
WARN_STDFUNC_OVERLOAD	warn if any user defined function has the same name as common system defined functions
WARN_VAR_NOTINIT	warn if a variable that may not be initialized along some path is used.Also warn if a variable not initialized along any path is used.
CHK_PATH_WITH_NORETURN	warn if a function that returns a non-void value has no return statement.Also warn if such a function may not have a return statement along certain paths.

Input constraints supported by checkada

The following constraints(with meanings alongside) may be specified in the input file.

constraint name	meaning

ASSO_NAMES_WITH_LOOPS	associate names with loops
ASSO_NAMES_WITH_BLOCKS	associate names with blocks
USE_LNAMES_ONALL_EXIT_STMTS	use the loop name on all exit statements from nested loops
INC_SMNAME_ATEND_PKGSPEC	include the simple name at the end of package specification and body.
INC_SMNAME_ATEND_TASKSPEC	include the simple name at the end of a task specification and body
INC_DESIG_ATEND_SUBBODY	include designator at the end of subprogram body
USE_NAMEASSOC_INS_GENE	use named association when instantiating generics
SHOW_MODEINDIC_OFALL_PROGEN_PARAMS	show mode indication of all procedure and entry parameters
USE_RANGECONS_ON_SUBTYPES	use range constraints on all subtypes
AVOID_ANON_TYPES	avoid anonymous types
INITALL_IN_RECORD	initialize all components within a record
DEALLOC_EXPLICITLY	deallocate all access variables explicitly

RESTRICT_REALS_IN_COMPS	restrict real operands if the operator is '='
ALWAYS_OTHERS_IN_CASE	always use 'others' choice in 'case' statement
NO_RANGE_ENUM_INCASE	do not use ranges of enumeration literals in 'case' statements
AVOID_EXIT_WHILE_FOR	avoid exit statements in 'while' and 'for' loops
AVOID_GOTO	avoid 'goto' statements
INIT_BEFORE_USE	initialize all variables before use
NOREF_NONLOCS_IN_GUARD	do not reference non-local variables in a guard
AVOID_ABORT	avoid 'abort' statement
USE_PARAMLESS_MAIN	use parameterless procedure as the main program
AVOID_MCCODE_INSERTS	avoid machine-code inserts
AVOID_IFACE_OTHER_LANGS	avoid interfacing Ada to other languages
AVOID_STD_OVERLOAD	do not overload any of the typenames in the package STANDARD
AVOID_ENUMLIT_OVERLOAD	avoid overloading enumeration literals

SET_MAX_PARAMS	set maximum number of parameters for any function or procedure
AVOID_NESTED_PKGSPECS	avoid nested package specifications
AVOID_NESTED_TASKINPKGSPECS	avoid task specifications in package specifications
SEPARATE_PKG_SPECBODY	put package specification and body in different files
USE_TYPEEQUALS	use type qualifiers rather than type casting
AVOID_ANON_TASKS	avoid anonymous tasks
AVOID_COND_ENTRY	avoid conditional entry calls
AVOID_SELWAIT_ELESE	avoid selective wait with 'else' parts
AVOID_TIMED_ENTRY	avoid timed entry calls
AVOID_SELWAIT_DELAY	avoid selective wait with delay alternatives
INIT_AT_DECL	initialize all variables at declaration
AVOID_VARS_IN_PKG_SPECS	avoid declaring variables in package specifications
NO_ENUMS_ARRAY_INDEX	prevent enumerated variables from indexing arrays

AVOID_RAISE_PREDEF_EXCEP	avoid explicitly raising predefined exceptions
USE_NAMEASSOC_SUBPROG_CALL	use named association in subprogram calls
AVOID_UNCHKD_CONV	avoid unchecked conversion
CATCH_MEMLOSS	catch unreferenced memory creation
AVOID_DANGLING_REF	do not create dangling references
AVOID_UNCHKD_DEALLOC	avoid using unchecked deallocation
RETURN_ALLPATHS	check if function has return on all paths
CATCH_UNUSED_VARS	warn about unused variables
OUT_PARAM_ASSIGNED	check that 'out' parameter is assigned prior to return

Using the tool

A usage session would go along the following lines

1. Create the input file either manually or with `makeinput` and specify what constraints are to be checked
2. Run the input file through `makestd` to create the output file
3. Compile and link the output file to the object files of the appropriate constraint-checker

4. Run the file to be tested through the constraint-checker to have the constraints checked. Warnings will be generated for any violations

It is assumed that a valid and compilable program would be submitted to this software.

Files

makestd is located in `code/makestd` of the distribution. It is composed of the following files:

`makestd.lex`: flex specification for lexer to read the constraint selection file.

`ada.std.decls`: declaration of the structure to represent the Ada coding-standard.

`c.std.decls`: declaration of the structure to represent the C coding standard.

Example

An example of an input file for C language is given below. In this example, all the 37 constraints have been turned on for enforcement.

```
LANGUAGE = c
NOT_ASSIGN_CONDTEST = yes
NO_PTRARITH_INEXPR = yes
NO_PTRARITH_INCONDTEST = yes
NOT_ALLOW_CHARINDEX = yes
NO_FLDB_INCONDEXP = yes
SET_EXTNAMESIGLENGTH = 5
NOT_REUSE_SCOPE_NAME = yes
SET_INTNAMESIGLENGTH = 5
NO_EXPR_SIDEFFS = yes
```



```

CHK_MISSING_ELSE = yes
CHK_DEEP_BREAKS = yes
CHK_BODYLESS_STMT = yes
CHK_BODY_NOT_BLOCK = yes
SET_MAXDEPTH_NESTING = 3
ENFORCE_ALL_NONEXTERN_STATIC = yes
SET_MAX_INCLUDE_DEPTH = 3
CHK_PRAGMA_INDENTED = yes
CHK_HEADER_OVERLOAD = yes
SET_MAXFIELD_STRUCT = 6
SET_MAXMEM_ENUM = 6
CHK_LOSSY_ASSIGN = yes
NO_SGN_UNSGN_ASSIGN = yes
SET_MAXLENGTH_STRLITERAL = 20
FLAG_SENSITIVE_LOOPS = yes
CHK_MULTIPLE_INCLUDE = yes
CHK_SIMILAR_NAMES = yes
CHK_MISSING_BREAK = yes
NOT_ALLOW_ENUMINDEX = yes
CHK_STORAGE_USED_AFTER_FREE = yes
CHK_STORAGE_NOT_FREED = yes
NO_CONCATENATION_OP = yes
NO_TOKEN_SUBSTITUTION = yes
VARS_INIT_AT_DECL = yes
CHK_RETURNOF_PTRTO_LOCAL = yes
WARN_STDFUNC_OVERLOAD = yes
WARN_VAR_NOTINIT = yes
CHK_PATH_WITH_NORETURN = yes
END

```

The output file is to be compiled into object form and linked with the object files of `checkc`. A call to the function `set_struct_c()` is made from the main file in `checkc`

prior to doing any checks in order to set the structure fields properly. Similarly for `checkada`, except that in this case the function to be called is `set_struct_ada()`.

Name

`makeinput` menu-based program to specify a coding standard

Synopsis

`makeinput [-lo] [file]`

This is a simple program that obviates the need for the user to type out the constraint selection file for `makestd`. Invoking this program starts a dialogue. The user first selects the language and then selects what constraints to turn on through yes/no replies. The dialogue generates the constraint selection file that is the input to `makestd`.

Flags

`-l` lists all available constraints for C and/or Ada

`-o <file>` interactively creates input file for `makestd` and prints it in `<file>`

Description

Not necessary

Files

`makeinput` is located in `code/makestd` of the distribution. It is composed of the following files:

`makeinput.h`: has a list of constraints and their mappings to symbolic names used in the constraint specification file.

`makeinput.c`: handles user-dialogue and generates code.

Example

Not necessary. The program is very simple.

Name

`checkc` coding-standard enforcer for C programs

Synopsis

`checkc <filename>`

`checkc` accepts the `<filename>` as input and produces as output the source code of the input file annotated with warnings enclosed in C comments at places where the set coding constraints are violated. Some source restructuring is also done to remove certain syntax constraint violations. The input file must be a compilable and pre-processed C file. `checkc` enforces upto 37 constraints. A program called `prepass` bundled with `checkc` checks some pre-processor constraints.

Flags

no flags

Description

`checkc` parses the C input file, builds a parse-tree and enforces constraints as it does a depth-first left-to-right traversal of the parse-tree. `checkc` is based on a C parsing package `ctree` made by Shawn Flisakowski. For `checkc` to be used, first the user must specify the constraints to be checked and create the coding-standard file by using `makeinput` and `makestd`. This file is then compiled to object form and linked with the rest of the object files for `checkc` to get the final executable.

Files

`checkc` is located in `code/checkc` of the distribution. It is composed of the following source files:

`prnt_tree.c`: has the code that traverses the parse-tree and enforces most of the constraints.

`syntab.c`: has functions to initialize and insert symbols into the symbol table.

`token.c`: has the function that returns the lexemes for parse-tree tokens. Used for unparsing the source code

`tree.c`: has functions to allocate and free parse-tree nodes. Also has a debugging function to inspect the parse-tree structure.

`tree_stk.c`: has functions to parse a series of files specified in the command line. Not of much use since now `checkc` only accepts one file.

`typetab.c`: has code that links type-table nodes to symbol-table nodes and to parse-tree nodes. Also code to enforce a couple of checks.

`test.output.c`: the coding standard file produced by `makestd`. Can be any other name but `makefile` has to be changed appropriately.

`extname.c`: declarations for the lists of internal and external names maintained by the program.

`Lexer.l`: lexical analyzer specification for flex.

`Gram.y`: parser specification file for bison.

`ctree.c`: main startup file.

`Lexer.h`, `ctree.h`, `prnt_tree.h`, `syntab.h`, `token.h`, `tree.h`, `typetab.h`: these are all headers for the files described above.

tree_stk.h: unused file

std.h: has the declaration of the coding-standard structure for C.

syshdrlist.h: has lists of common system-headers and common system functions used to check overloading by user.

Example

For the case where all the 37 possible constraints are turned on(see makestd), let the input file be

```
main()
{
    int *a;
    float arr[100];
    char c;
    double abcdef,abcdeg;
    unsigned int avar;
    signed int bvar=10;
    int simi;
    float siml;
    char siml;
    int au;

    struct tag {
        int a; float d; int c;
        int b; float e; float f;
        char g;
    } sname;

    bvar = -10;

    a = (int *)malloc(sizeof(int));
```

```

*a = 20;

if (i=0) {
    printf("a statement about the program\n");
}

if ((*a+1)==31) {
    printf("another statement %d \n",*a+1);
    c = 'a';
    arr[c] = 10.9;
}

while (arr[10] < 100.0) {
    printf("a=%d \n",*(a++));

    while (arr[12] >90.1) {
        while (1) {
            if (1) {
                char simi;
                simi = 10;
            }
            c = simi;
        }
        break;
    }
    avar = bvar;
}

if (1);

if (1) printf("a statement");
while (1) printf("a statement");

```



```

switch (au) {
case 10: break;
case 11: printf("printf statement");
default: break;
}

```

```

evar = C;
arr[ evar ] = 18.00;

```

```

{
    char *ptr;
    ptr = (char *)malloc(10);
    {
        char *tptr;
        free(ptr);
        tptr = (char *)malloc(80);
    }
    *(ptr+2) = 'k';
}
}

```

Applying checkc produces the following output

```

/* warning:[report.c:5]: enum has more members(8) than permitted(6) */

/* warning:[report.c:24]: struct has more number of fields(7) than
    permitted(6) */

/* warning:[report.c:46]:max allowed nesting level exceeded */

/* warning:[report.c:66]:maybe missing <break> in prior case stmt */
int i
/* warning: variable i not initialized at declaration */

```

```

/* warning:non-extern global var:i is not static */
, j
/* warning: variable j not initialized at declaration */

/* warning:non-extern global var:j is not static */
;
extern int uuuvcd
/* warning: variable uuuvcd not initialized at declaration */
, uuuvcei
/* warning: variable uuuvcei not initialized at declaration */
;
static int au=0;

enum etag { A , B , C , D , E , F , G , H } evar;
main()
{
int *a
/* warning: variable a not initialized at declaration */
;
float arr[100];
char c
/* warning: variable c not initialized at declaration */
;
double abcdef
/* warning: variable abcdef not initialized at declaration */
, abcdeg
/* warning: variable abcdeg not initialized at declaration */
;
unsigned int avar
/* warning: variable avar not initialized at declaration */
;
signed int bvar=10;
int simi

```

```

/* warning: variable simi not initialized at declaration */
;
float siml
/* warning: variable siml not initialized at declaration */
;
char sim1
/* warning: variable sim1 not initialized at declaration */
;
int au
/* warning: variable au not initialized at declaration */
;

struct tag { int a ;
float d ;
int c ;
int b ;
float e ;
float f ;
char g; } sname;
bvar=(-10);
a= ( int * ) malloc( sizeof(int ) );
*a=20;
if (i=
/* warning:assignment is part of conditional test */
0){
{
printf( "a statement about the program\n"
/* warning: length of string literal(30)exceeds the maximum
permitted(20) */
);
}

/* warning:prior if statement has missing 'else' clause at the end */

```

```

;}
if (((*a
/* warning:ptr arithmetic in conditional test */.
+1)==31)){
{
printf( "another statement %d \n"
/* warning: length of string literal(22)exceeds the maximum
permitted(20) */
,(*a+1) );
c=97;
arr[c
/* warning:char used to index array */
]=10.900000;
;
}

/* warning:prior if statement has missing 'else' clause at the
end */
;}
while ((arr
/* warning:float used in conditonal test */
[10]<100.000000)){
{
printf( "a=%d \n" ,*(a++) );
while ((arr
/* warning:float used in conditonal test */
[12]>90.100000)){
{
while (1){
{
if (1){
{
char simi

```

```

/* warning: variable simi not initialized at declaration */
;
simi=10;
}

/* warning:prior if statement has missing 'else' clause at the
    end */
;}
;
c=
/* warning: unassigned variable simi is used */
simi
/* warning:loss making assignment */
;
;
}
;}
;
break
/* warning:'break' statement inside multiple loop-nest */
;
;
}
;}
avar=bvar
/* warning:signed assigned to unsigned */
;
;
}
;}
if (1){

/* warning:'if' statement has no body */

```

```

/* warning:'if' stmt body is not a block */

/* warning:prior if statement has missing 'else' clause at the
    end */
;}
if (1){
printf( "a statement" )
/* warning:'if' stmt body is not a block */

/* warning:prior if statement has missing 'else' clause at the
    end */
;}
while (1){
printf( "a statement" )
/* warning:'while' stmt body is not block */
;}

switch ( au ) {
case (10) :
break ;
case (11) :
printf( "printf statement" );
default :
break ;
;
}
evar=C;
arr[evar
/* warning:enum used to index array */
]=18.000000;
{
char *ptr

```

```

/* warning: variable ptr not initialized at declaration */
;
ptr= ( char * ) malloc( 10 );
{
char *tptr
/* warning: variable tptr not initialized at declaration */
;
free( ptr );
tptr= ( char * ) malloc( 80 );
;
}

/* warning:pointer [tptr] not freed before scope exit */
*(ptr
/* warning: storage ptr used after freeing */
+2)=107;
;
}
;
}

/* warning:pointer [a] not freed before scope exit */

/* warning:similar looking names[report.c:12:abcdef][report.c:12:abcdeg]
at nesting levels 1 and 1 */

/* warning:similar looking names[report.c:13:avar][report.c:14:bvar]
at nesting levels 1 and 1 */

/* warning:similar looking names[report.c:16:siml][report.c:47:simi]
at nesting levels 1 and 5 */

```

```

/* warning:similar looking names[report.c:16:siml][report.c:15:simi]
   at nesting levels 1 and 1 */

/* warning:similar looking names[report.c:16:siml][report.c:17:sim1]
   at nesting levels 1 and 1 */

/* warning:similar looking names[report.c:47:simi][report.c:17:sim1]
   at nesting levels 5 and 1 */

/* warning:similar looking names[report.c:15:simi][report.c:17:sim1]
   at nesting levels 1 and 1 */

/* warning:symbol [report.c:47][report.c:15] simi is reused in different
   scopes at nesting levels 5 and 1 */

/* warning:symbol [report.c:18][report.c:4] au is reused in different
   scopes at nesting levels 1 and 0 */

/* warning:[report.c:3:uuuvcd] [report.c:3:uuuvcei] external names are
   not significant in 5 chars */

/* warning:[report.c:4:au] [report.c:18:au] internal names are not
   significant in 5 chars */

/* warning:[report.c:12:abcdef] [report.c:12:abcdeg] internal names
   are not significant in 5 chars */

/* warning:[report.c:15:simi] [report.c:47:simi] internal names are
   not significant in 5 chars */

```

As can be seen, the code is re-created and annotated with at the earliest point where violations of the coding- standard are detected.

Name

`checkada` coding-standard enforcer for Ada programs

Synopsis

```
checkada < file1 > [file2]:...[filen]
```

`checkada` accepts valid Ada program(s) as input and checks the code for conformance to a set Ada coding-standard. The output consists of the input code marked with line-numbers on each line. This is followed by the warnings for any violations of the coding constraints. Each warning has a line-number to indicate where in the source file that violation was found. In case more than one file is given, they are all treated as one single large file. Upto 47 constraints can be enforced by `checkada`.

Flags

No flags

Description

The actions of `checkada` are similar to those of `checkc` as far as parsing the input Ada code and building the parse-tree are concerned. For `checkada` to be used, a coding-standard file should also be created with `makeinput` and `makestd` and linked with the rest of the `checkada` object files. This tool enforces 47 constraints, all selected from the book *Ada quality and style: guidelines for professional programmers* by the Software Productivity Consortium (1989, New York: Van Nostrand Reinhold).

Files

`checkada` is located in `code/checkada` of the distribution. It is composed of the following files:

`ada_cons.c`: the coding-standard file produced by `makestd`. Can be any other name but `makefile` has to be changed

`main.c`: startup file

`show_tree.c`: contains code to print out the code by traversing the parse-tree. All the constraint checks are also done here.

`table.c`: has functions to insert records into type and symbol tables, symbol lookup and table storage recovery

`tree_node.c`: functions to create leaf and interior nodes in the parse-tree and free the tree after use.

`warn_mesg.c`: has function to print warning messages based on the type of constraint violated.

`ada_cons.h`: has declarations of coding-standard structure for Ada.

`ada_lexer.h`: only used for including `tree_node.h`

`name_stack.h`: structure used to keep track of subprogram and package name context.

`table.h`: declarations for symbol and typetable records and table manipulation functions for both tables.

`tree_node.h`: declaration for the tree-node type structure.

`warn_mesg.h`: macro definitions for indexing warning messages in `warn_mesg.c`.

`ada_lexer.l`: lexical analyzer specification for Ada. This is the input to `flex`.

ada_parser.y: Ada parser specification for bison.

Example

By default the output comes on the screen. Output may be re-directed to a file if desired. A simple example is given below for the case where all checks except *avoid anonymous types* are turned on.

```
Procedure main is
x: float;
r: integer;
begin
  declare
    x: integer;
    r : float;
  begin
    x := 2;
    x := r;
    r := 10.1;
  end;

  declare
    type Z is (A,B,C);
    x: Z;
  begin
    x := r;

  select
    when r=10 => accept T1;
  or
    when r=20 => accept T2;
  end select;
```

```

    end;
x := 10.2;
x := r;
x := float(r);
free(r);
end;

```

The output produced is

```

[2]  Procedure main is
[3]  x: float;
[4]  r: integer;
[5]  begin
[6]      declare
[7]      x:integer;
[8]      r : float;
[9]      begin
[10]          x := 2;
[11]          x := r;
[12]          r := 10.1;
[13]      end;
[14]
[15]      declare
[16]      type Z is (A,B,C);
[17]      x:Z;
[18]      begin
[19]          x := r;
[20]
[21]      select
[22]          when r=10 => accept T1;
[23]      or
[24]          when r=20 => accept T2;
[25]      end select;
[26]

```

```
[27]      end;
[28]  x := 10.2;
[29]  x := r;
[30]  x := float(r);
[31]  free(r);
[32]  end;
[33]
Successful parse.....
```

Warning [32] Should include designator at end of subprogram body

Warning [3] Variable(s) should be initialized at declaration

Warning [4] Variable(s) should be initialized at declaration

Warning [6] Should associate name with block

Warning [7] Variable(s) should be initialized at declaration

Warning [8] Variable(s) should be initialized at declaration

Warning [11] variable 'r' not initialized prior to use

Warning [15] Should associate name with block

Warning [17] Variable(s) should be initialized at declaration

Warning [19] variable 'r' not initialized prior to use

Warning [22] variable 'r' not initialized prior to use

Warning [22] non-local 'r' referenced in guard

Warning [24] variable 'r' not initialized prior to use

Warning [24] non-local 'r' referenced in guard

Warning [29] variable 'r' not initialized prior to use

Warning [30] Use type-qualifiers rather than type-casting
whenever possible

Warning [30] variable 'r' not initialized prior to use

Warning [31] Named association not used in subprogram call

Freeing parse tree....

freeing symbol table.....

freeing type table.....

Name

`createparse` generates a skeleton front-end

Synopsis

`createparse <specification_file>`

`createparse` accepts a specification file for any language grammar as input and generates the files for a basic front-end. This includes a skeleton lexer(with blank actions to be filled in by the user),a parser specification that also has code to build the parse-tree during parsing,and a function to traverse the parse-tree and print out the source code again.This tool is useful to create the starting point from where constraint-checkers, beautifiers and other source-to-source translators can be built.

Flags

no flags

Description

Format of the specification file

`createparse` reads the specification file supplied by the user and creates certain files.This file lists some information about the grammar of the language for which a front-end is to be produced.The format is given below

```
%{{tokens
  <token_1>  <corresponding_lexeme>
  <token_2>  <corresponding_lexeme>
  .
  .
  .
  <token_n>  <corresponding_lexeme>
%}}
```

```
%{{lex_subs
  <short_form_1>  <corresponding_regular_expression>
  <short_form_2>  <corresponding_regular_expression>
  .
  .
  .
  <short_form_n>  <corresponding_regular_expression>
%}}
```

```
%{{lang_keywords_tokens
  <token_1>      <corresponding_keyword>
  <token_2>      <corresponding_keyword>
  .
  .
  .
  <token_n>      <corresponding_keyword>
%}}
```

```
%{{max_numright_children  <number>
%}}
```

```
%{{grammar
```



```
<bison grammar specification here>
%}}
```

The *tokens* section has the lexical tokens and their associated lexemes. The lexeme part may also be a regular expression. The *lex_subs* section has the short names of regular expressions that are very often used in lexical analyzers to save typing and their associated regular expressions. The *lang_keywords_tokens* section lists the keywords of the language and their corresponding tokens returned to the parser. The *max_numright_children* section gives the maximum number of elements on the right-hand-side of a production among all the productions in the grammar. This information is necessary for correctly generating the structure definition for the parse-tree nodes and the correct actions in the bison specification file. The *grammar* section lists the context-free grammar of the language (without any action parts).

createparse reads the language specification file and generates certain files. The bison specification itself is parsed and the bison input file produced by a separate program called **genbison** which is called by **createparse**. Before using the generated tool, first the full lexer has to be created from the generated skeleton by adding the actions, then the tool can be compiled by using the makefile named **auto_makefile** that is also generated. The files generated are **auto_parser.y**, **auto_tree_node.h**, **auto_tree_node.c**, **auto_main.c**, **auto_show_tree.c**, **auto_lexer.l** and **auto_makefile**.

Files

createparse is located in **code/auto/main** of the distribution. It is composed of the following source files:

read_spec.l: this is the lexer to read the specifications and also generates most of the files.

genbison which is used by **createparse** to create the bison specification file is located in **code/auto/genbison**. It consists of the following files:

gb_lexer.l: lexer to read the grammar specifications

`gb_main.c`: main file

`gb_parser.y`: parses the grammar specifications and generates appropriate action parts

`gb_tree_node.h`: has the declaration for the type of *yylval*

Example

Example specification files for C and Ada are bundled along with the tool in `code/auto/input` as `c_input.spec` and `ada_input.spec`

Bibliography

- [1] Aho,A.V.,Sethi,R.,& Ullman,J.D.(1986). *Compilers:Principles,Techniques and Tools*. Reading,MA:Addison-Wesley.
- [2] Holub,Allen.I.(1990). *Compiler design in C*. Prentice Hall India.
- [3] The software productivity consortium.(1989). *Ada quality and style:guidelines for professional programmers*. New York:Van Nostrand Reinhold.
- [4] Amoroso,Serafino.,& Ingargiola,Giorgio.(1985). *Ada-an introduction to program design and coding*. Howard W. Sams&Co.
- [5] Reps,Thomas.W.,& Teitelbaum,Tim.(1988). *The synthesizer generator.A system for constructing language-based editors*. New York:Springer-Verlag.
- [6] Kernighan,B.W.,& Ritchie,D.M.(1988). *The C Programming Language*. Prentice Hall,Englewood Cliffs,N.J.
- [7] Evans,David.,Gutttag,John.,Horning,James.,& Tan,Yang.Meng.(1994, December). LCLint: a tool for using specifications to check code.*SIGSOFT symposium on the foundations of Software Engineering*.
Also [On-line]<ftp://larch.lcs.mit.edu/pub/Larch/lclint/fse94.ps.gz>
- [8] Painter,Jo.Ann.,& Lowe,Rick.(1995,July). *C style and coding standards for the SDM project*.
[On-line]http://www-c8.lanl.gov/sdm/DevelopmentEnv/SDM_C_Style_Guide.html

- [9] Cannon,L.W.,Elliott.R.A.,Kirchoff,L.W.,Miller,J.H.,Milner,J.M.,Mitze,
R.W.,Schan,E.P.,Whittington,N.O.,Spencer,Henry.,Keppel,David.,&Brader,
Mark.(1990,June). *Recommended C style and coding standards*.Technical re-
port,in the public domain.
[On-line]<http://www.apocalypse.org/pub/u/paul/docs/cstyle/cstyle.html>
- [10] *GNU coding standards*.
[On-line]http://www.ele.auckland.ac.nz/doc/misc/standards_toc.html
- [11] *STARS project reusability guidelines*.
[On-line][http://source.asset.com/WSRD/ASSET/A/209/elements/
c1550c.txt](http://source.asset.com/WSRD/ASSET/A/209/elements/c1550c.txt)
- [12] On-line manual for lint.
- [13] Dolenc,A.,Lemmke,A.,Keppel,D.,& Reilly,G.V.(1990,November,8th Revision).
Notes on writing portable programs in C.
[On-line]<ftp://cs.washington.edu/pub/cport.tar.Z>
- [14] Noyelle,Yves.L.Disciplined C.(1995,December). *ACM SIGPLAN Notices*,
Volume 30,N0.12
- [15] Evans,David.(1994,September). *LCLint User's Guide*,Version 1.4
[On-line]<ftp://larch.lcs.mit.edu/pub/Larch/lclint/lclint1.4.userguide.ps.Z>
- [16] Stallman,Richard.(1988,February). *GNU Emacs Manual*,Sixth Edition,Emacs
Version 18.
- [17] Donnelly,Charles.,& Stallman,Richard.(1990). *The Bison Manual*,Free Software
Foundation Inc.,Cambridge,Massachusetts.Available via anonymous ftp as part
of the Bison distribution at prep.ai.mit.edu
- [18] Paxson,Vern.(1990).On-line manual pages distributed with the Flex software
package.Available via anonymous ftp from prep.ai.mit.edu

- [19] Kokol, Peter., Žumer, Viljem., Brest, Janez., & Mernik, Marjan. (1995, May).
PROMIS: A software metrics tool generator. *ACM SIGPLAN Notices*, Volume
30, NO.5
- [20] *Knowledge Software Home Page*.
[On-line] <http://www.knosof.co.uk/>
- [21] *Ada Home: the Home of the brave Ada programmers*.
[On-line] <http://www.adahome.com/>
- [22] *Ada information clearinghouse FTP server*.
[On-line] <ftp://ajpo.sei.cmu.edu/>
- [23] *Grammatech Inc. Home Page*.
[On-line] <http://www.grammatech.com/>
- [24] *ctree version 2*.
[On-line] <ftp://ftp.cs.wisc.edu/coral/tmp/spf/ctree-0.2.tar.gz>